IA-64 Architecture Innovations

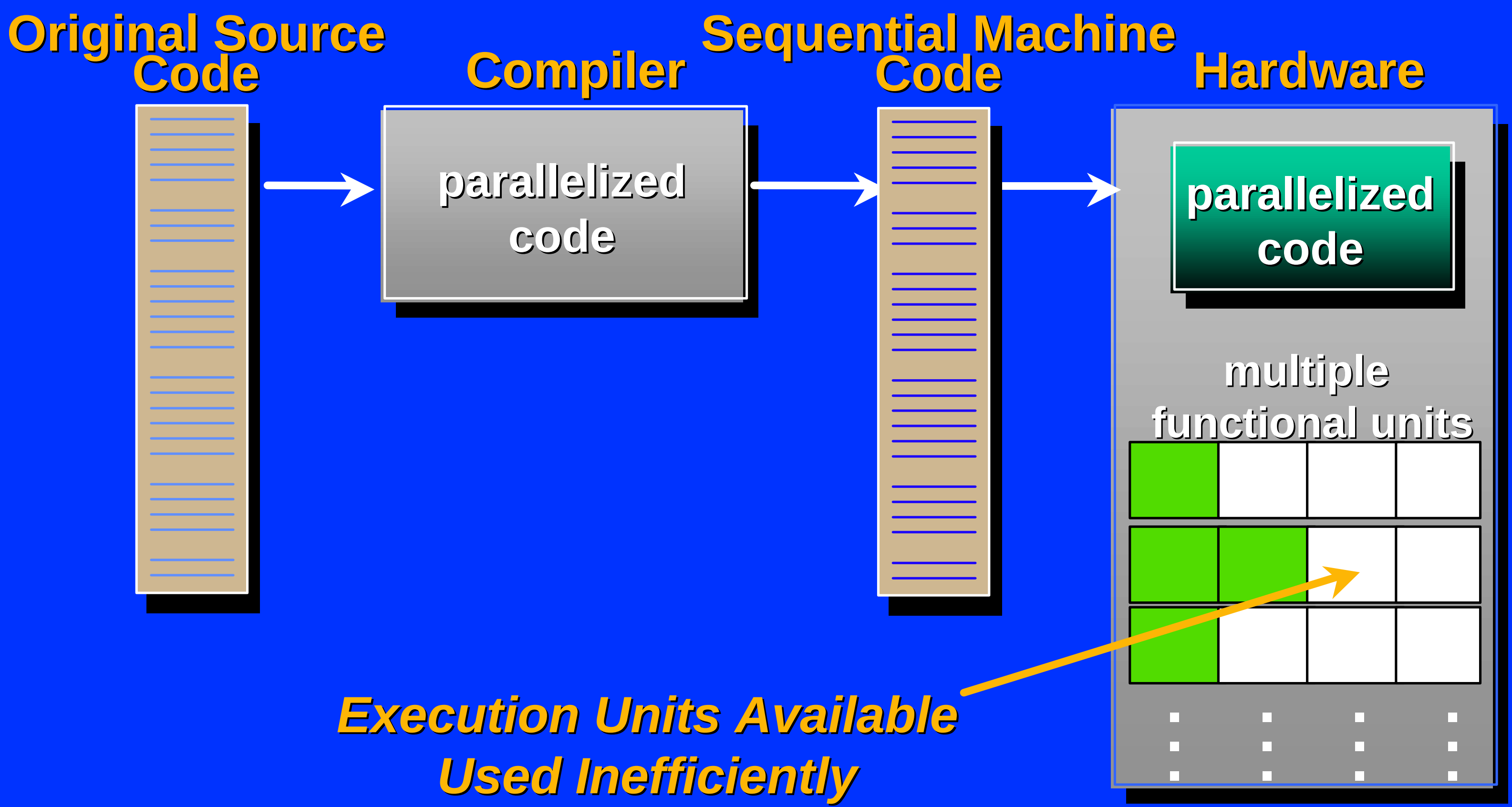
John Crawford
Architect & Intel Fellow
Intel Corporation

Jerry Huck
Manager & Lead Architect
Hewlett Packard Co.

Agenda

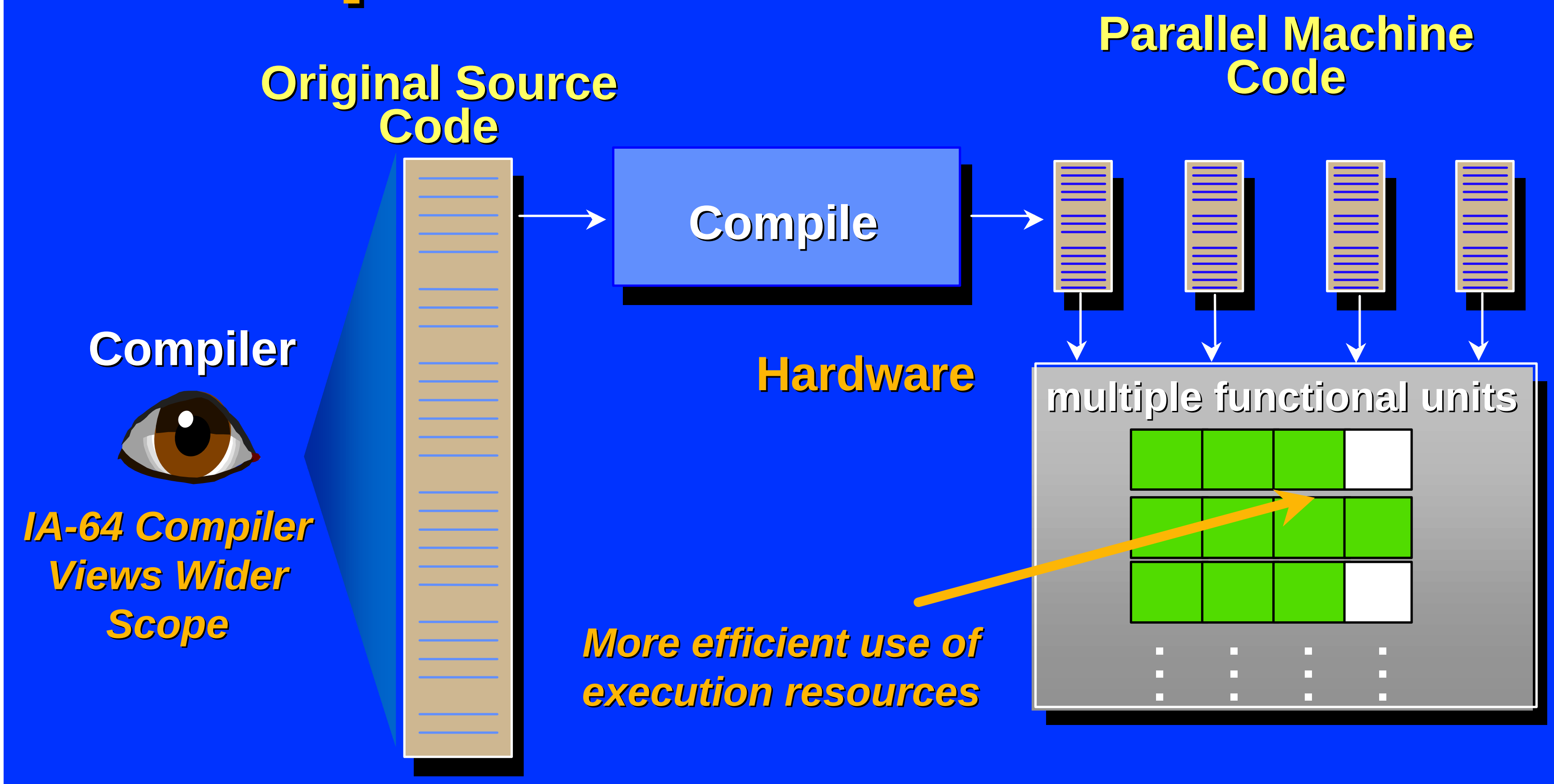
- Architecture Principles
- Predication & Speculation
- Branch Architecture
- Software Pipelining

Traditional Architectures: Limited Parallelism



Today's Processors often 60% Idle

IA-64 Architecture: Explicit Parallelism

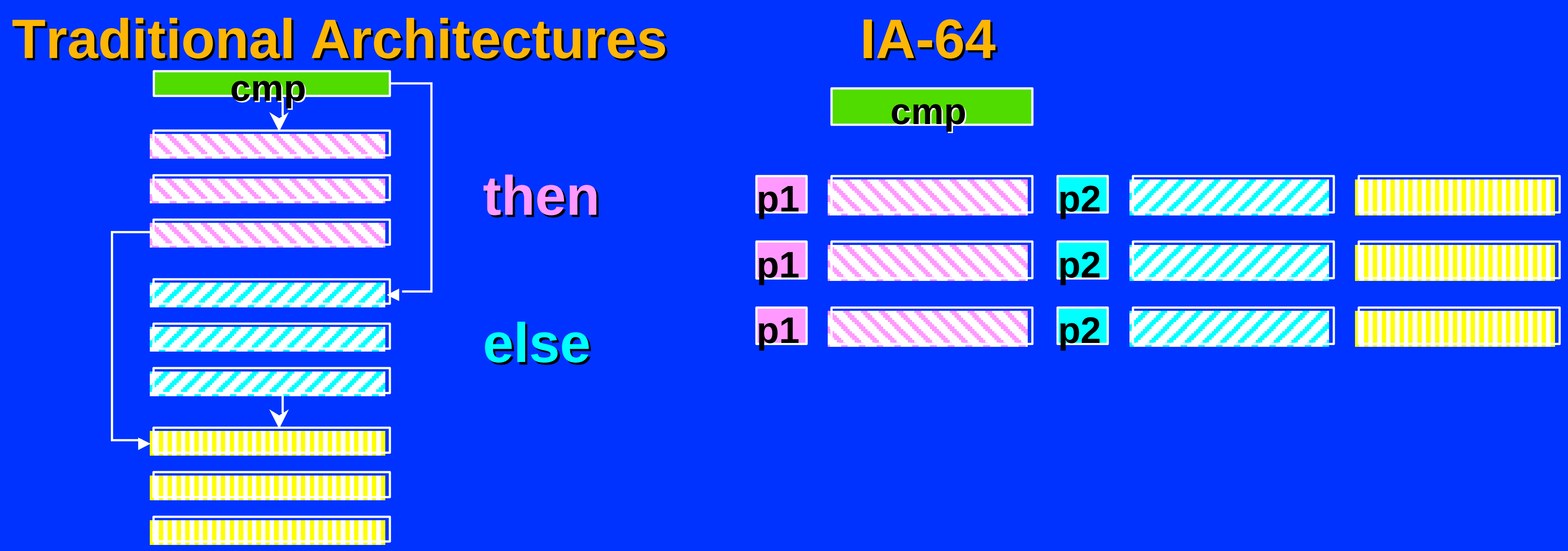


Increases Parallel Execution

IA-64 Principles

- **Explicitly parallel:**
  - Instruction level parallelism (ILP) in machine code
  - Compiler schedules across a wider scope
- **Enhanced ILP :**
  - Predication, Speculation, Software pipelining, ...
- **Fully compatible:**
  - Across all IA-64 family members
  - IA-32 in hardware and PA-RISC through instruction mapping
  - Inherently scalable
- **Massively resourced:**
  - Many registers
  - Many functional units

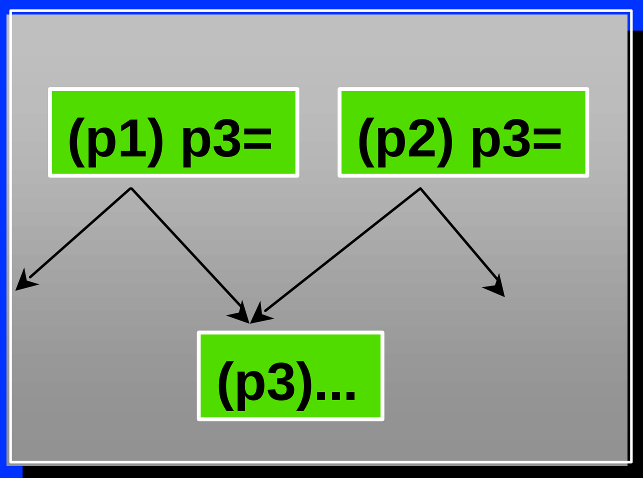
Predication



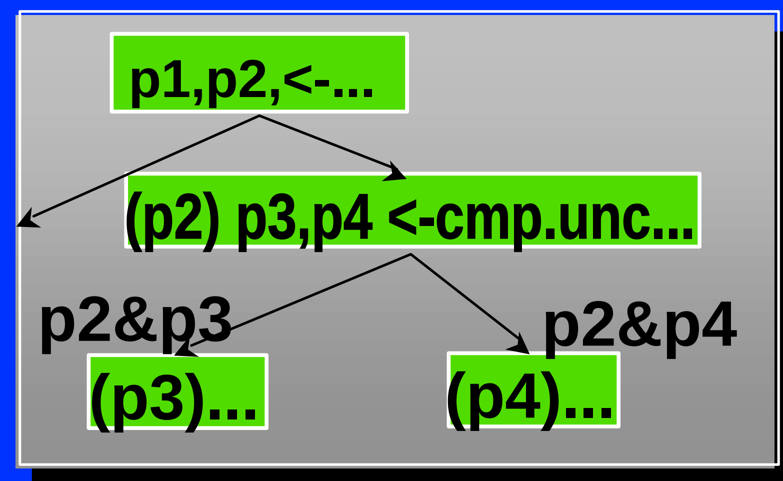
- Removes branches, converts to predicated execution
  – Executes multiple paths simultaneously
- Increases performance by exposing parallelism and reducing critical path
  – Better utilization of wider machines
  – Reduces mispredicted branches

Predication Review

- Two kinds of normal compares
  - Regular
  - Unconditional (nested IF’s)



Regular: p3 is set just once

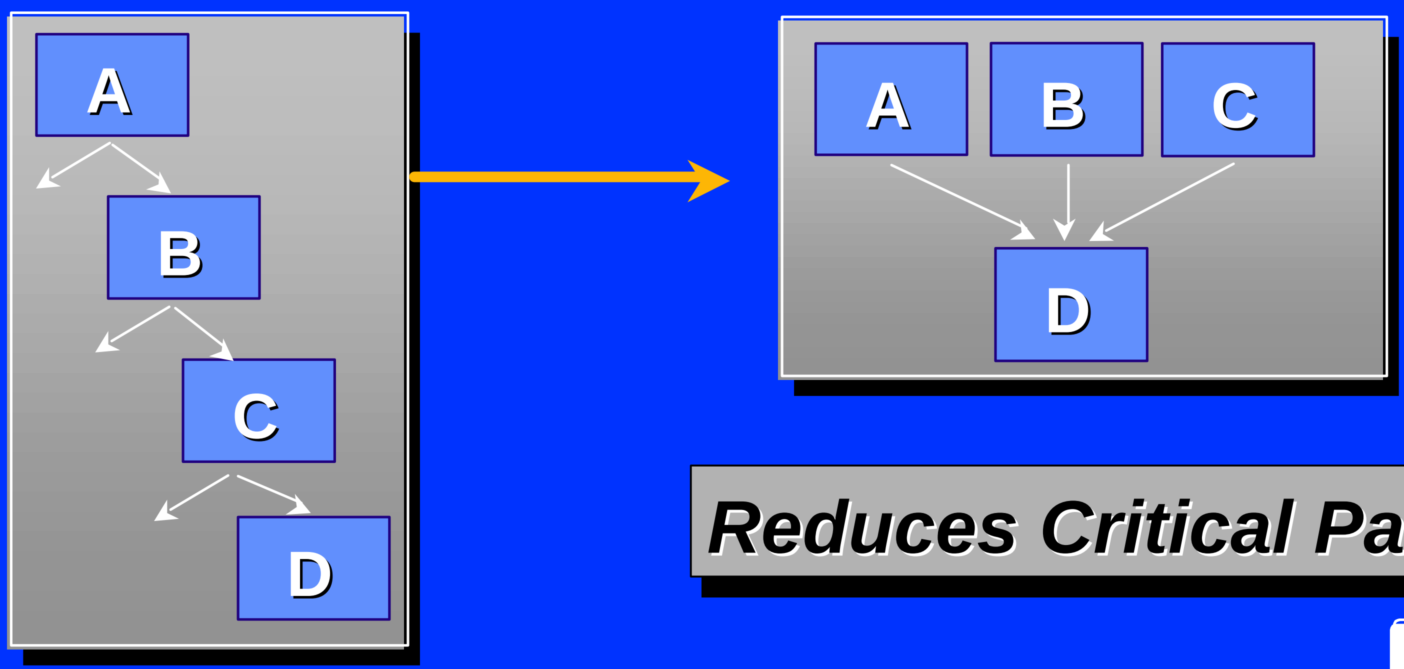


Unconditional: p3 and p4 are AND’ed with p2

Opportunity for Even More Parallelism

Introducing Parallel Compares

- Three new types of compares:
  - AND: both target predicates set FALSE if compare is false
  - OR: both target predicates set TRUE if compare is true
  - ANDOR: if true, sets one TRUE, sets other FALSE



Reduces Critical Path

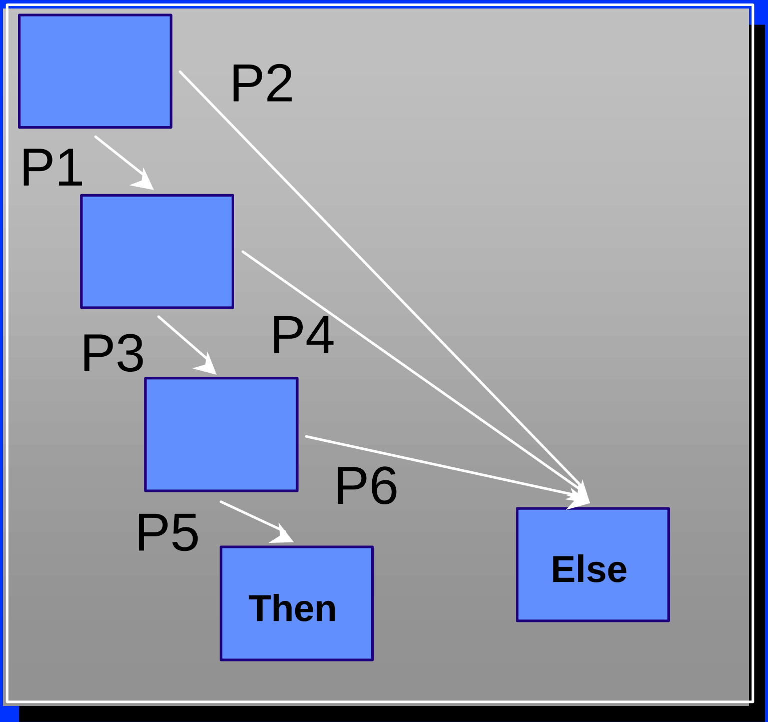
Eight Queens Example

if ((b[j] == true) && (a[i+j] == true) && (c[i-j+7] == true))

Unconditional Compares

1	R1=&b[j] R3=&a[i+j] R5=&c[i-j+7]
2	ld R2=[R1] ld~~.s~~ R4=[R3] ld~~.s~~ R6=[R5]
4	P1,P2 <-cmp.unc(R2==true)
5	~~(p1) chk.s R4~~ (p1) P3,P4 <-cmp.unc(R4==true)
6	~~(p3) chk.s R6~~ (p3) P5,P6 <-cmp.unc(R5==true)
7	(P5) br then else

8 queens control flow



intel®

HEWLETT PACKARD

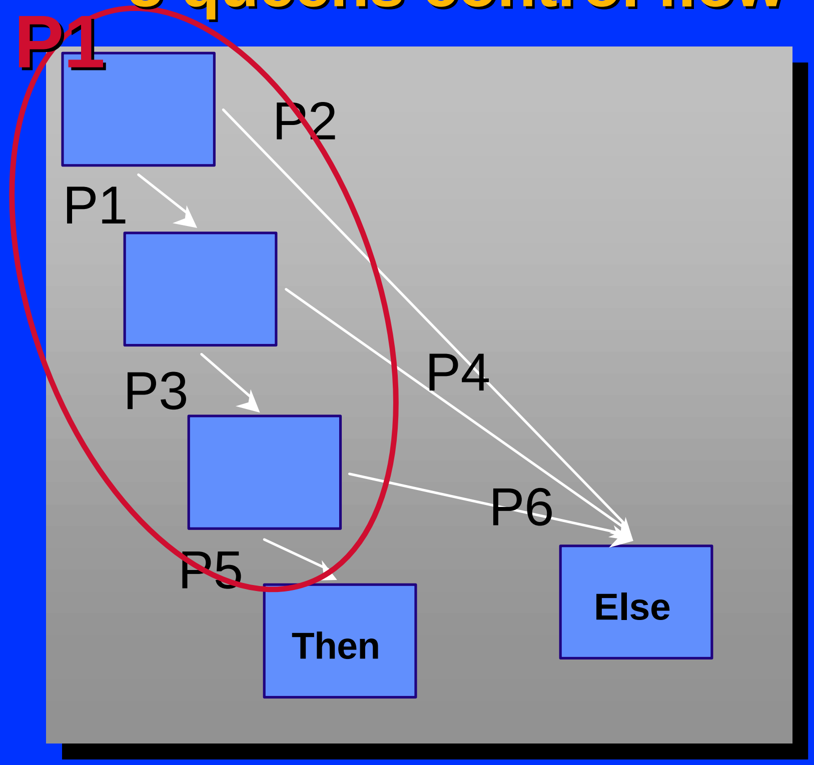
Eight Queens Example

if ((b[j] == true) && (a[i+j] == true) && (c[i-j+7] == true))

Parallel Compares

1	R1=&b[j] R3=&a[i+j] R5=&c[i-j+7] p1 <- true
2	ld R2=[R1] ld R4=[R3] ld R6=[R5]
4	p1,p2 <- cmp.and(R2==true) p1,p2 <- cmp.and(R4==true) p1,p2 <- cmp.and(R6==true)
5	(p1) br then
	else

8 queens control flow



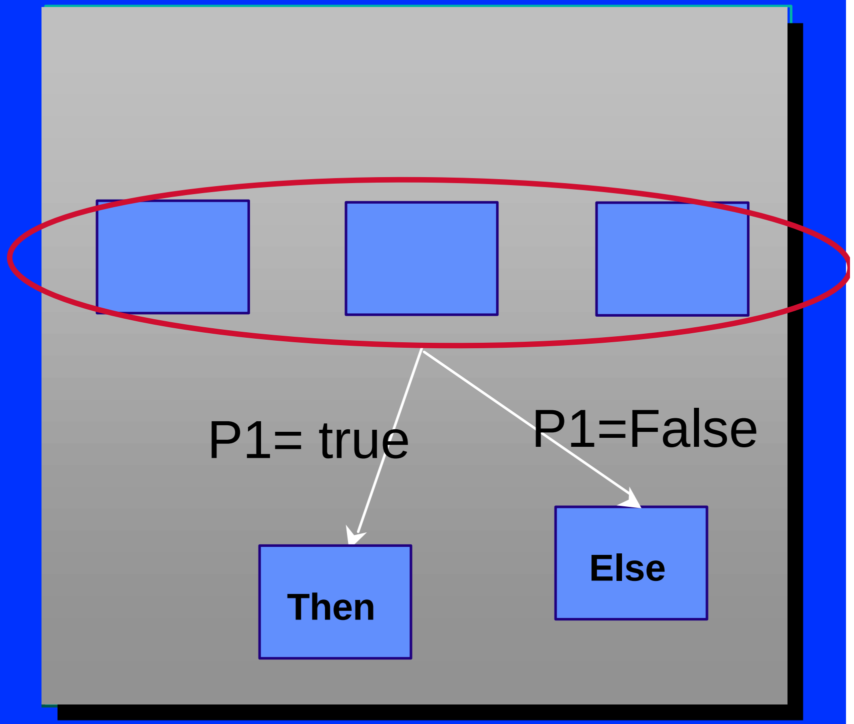
Eight Queens Example

if ((b[j] == true) && (a[i+j] == true) && (c[i-j+7] == true))

Parallel Compares

Cycle	Instructions
1	R1=&b[j] R3=&a[i+j] R5=&c[i-j+7] p1 <- true
2	ld R2=[R1] ld R4=[R3] ld R6=[R5]
4	p1,p2 <- cmp.and(R2==true) p1,p2 <- cmp.and(R4==true) p1,p2 <- cmp.and(R6==true)
5	(p1) br then else

8 queens control flow



Reduced from 7 cycles to 5

Five Predicate Compare Types

- (qp) p1,p2 <- cmp.relation
  - if(qp) {p1 = relation; p2 = !relation};
- (qp) p1,p2 <- cmp.relation.unc
  - p1 = qp&relation; p2 = qp&!relation;
- (qp) p1,p2 <- cmp.relation.and
  - if(qp & (relation==FALSE)) { p1=0; p2=0; }
- (qp) p1,p2 <- cmp.relation.or
  - if(qp & (relation==TRUE)) { p1=1; p2=1; }
- (qp) p1,p2 <- cmp.relation.or.andcm
  - if(qp & (relation==TRUE)) { p1=1; p2=0; }

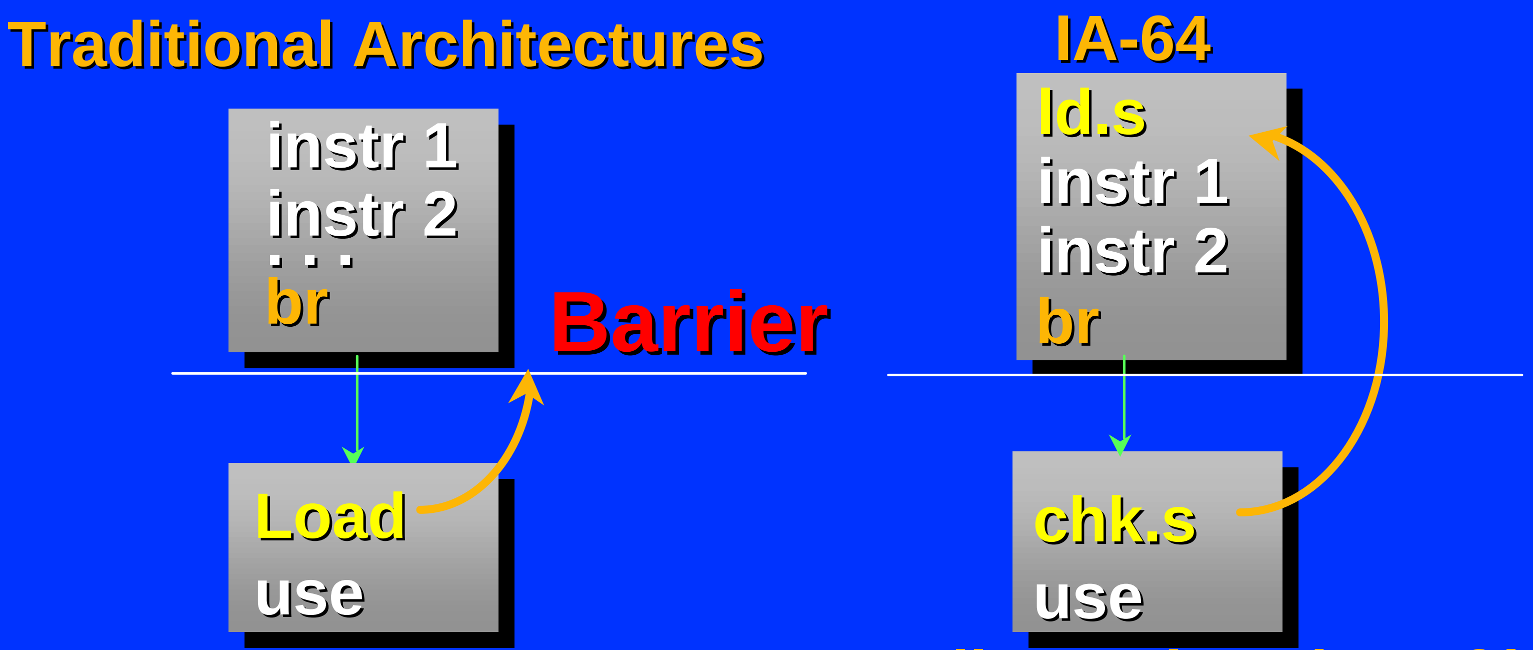
Tbit (Test Bit) Also Sets Predicates

Predication Benefits

- Reduces branches and mispredict penalties
  - 50% fewer branches and 37% faster code*
- Parallel compares further reduce critical paths
- Greatly improves code with hard to predict branches
  - Large server apps- capacity limited
  - Sorting, data mining- large database apps
  - Data compression
- Traditional architectures' "bolt-on" approach can't efficiently approximate predication
  - Cmove: 39% more instructions, 23% slower performance*
  - Instructions must all be speculative

* Source: S. Mahlke, 1995

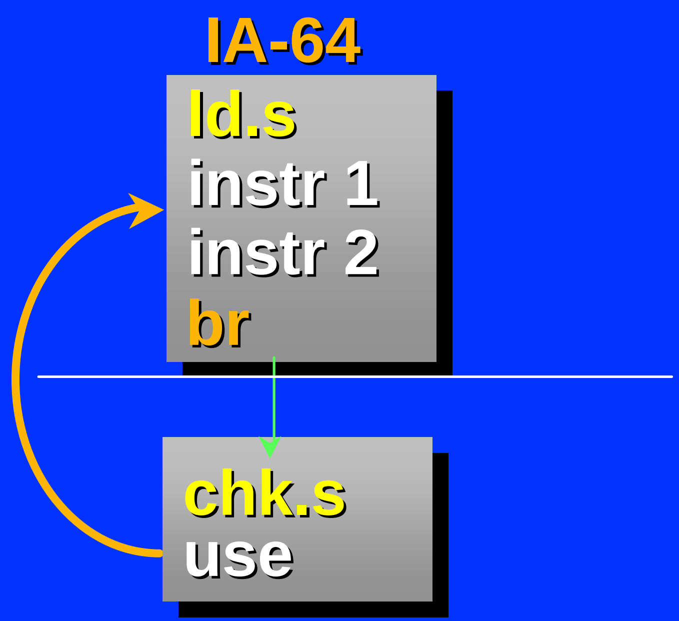
Speculation Review



Allows elevation of load, even above a branch

- Memory latency is a major performance bottleneck in today’s systems
  - CPU to memory gap increasing

Hoisting Uses



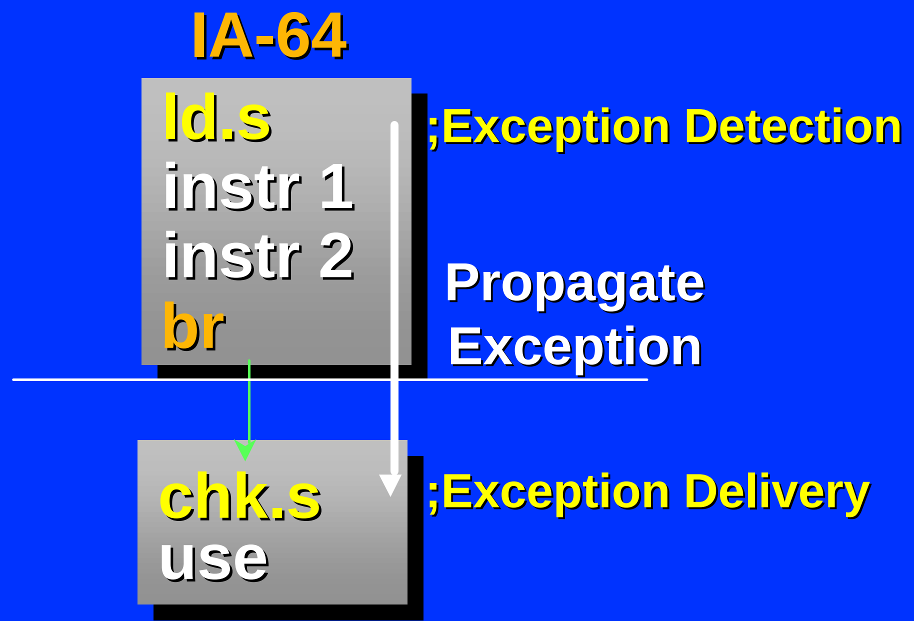
- The uses of speculative data can also be executed speculatively
  - distinguishes speculation from simple prefetch

Enables Further Parallelism

intel®

HEWLETT PACKARD

Introducing the NaT
("Not a Thing")



- NaT is the GR's 65th bit that indicates:
  - whether or not an exception has occurred
  - branch to fixup code required
- NaT set during ld.s, checked by Chk.s

intel®

HEWLETT PACKARD

Propagation

- All computation instructions propagate NaTs to reduce number of checks

```
ld8.s  r3 = (r9)
ld8.s  r4 = (r10)
add r6 = r3, r4
ld8.s  r5 = (r6)
p1,p2 = cmp(...)
```

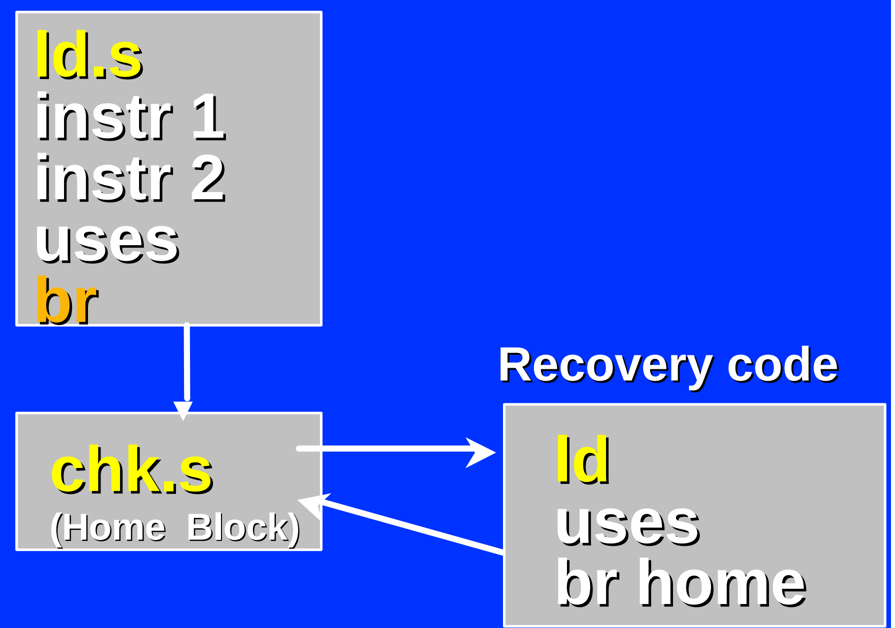
```
chk.s r5
sub r7 = r5,r2
```

Allows single chk on result

- Cmp propagates “false” when writing predicates
- RISC architectures require more instructions for equivalent integrity
  – e.g., non faulting load

Exception Deferral: More Than Skin Deep

- Deferral allows the efficient delay of costly exceptions
- OS controlled deferral by hardware of:
  - Page faults
  - Protection violations
  - …
- NaTs enable deferral with recovery
- Efficiently support structured exception handling in C/C++



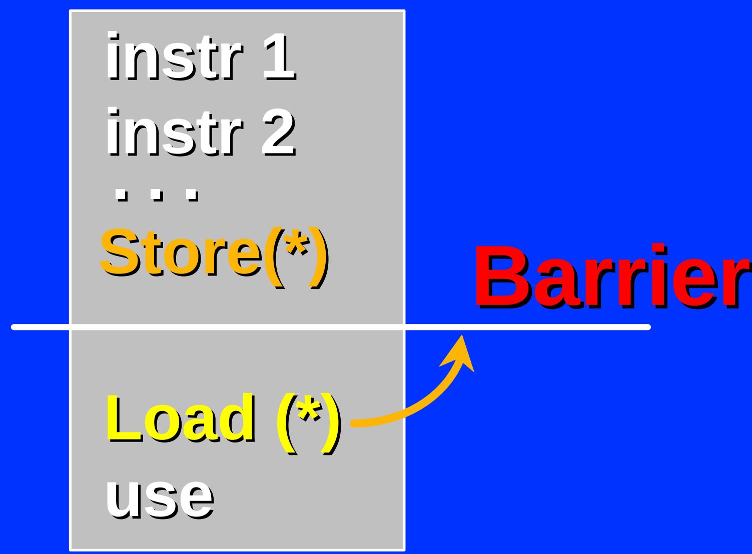
Complete Solution for Exception Management

Control Speculation Summary

- All loads have a speculative form that sets the NaT bit when deferring exceptions
- Computational instructions propagate NaTs
- OS controls deferral of faults but supported directly in HW - “no-fault speculation”
  - Minimizes overhead of data that is not used
- Chk more effective than non-faulting load

Store Barrier

Traditional Architectures



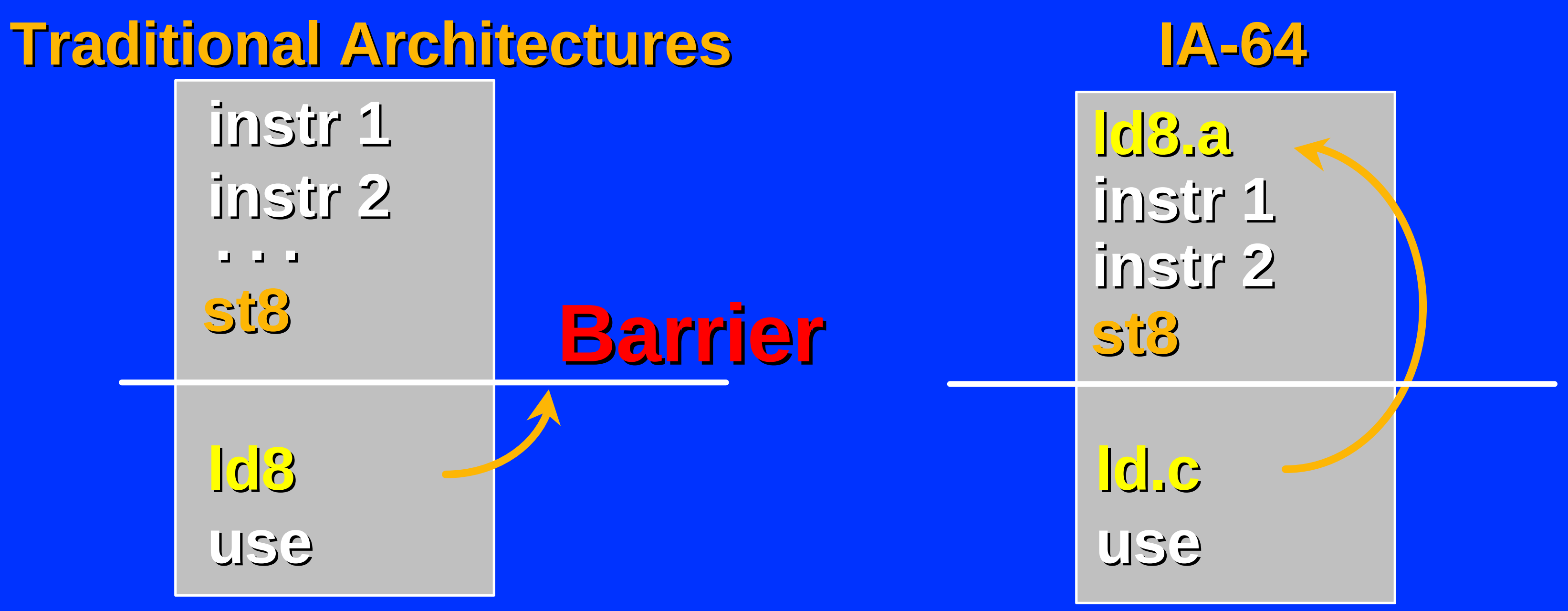
Traditional architectures limited by the Store Barrier

intel®

HEWLETT PACKARD

Introducing Data Speculation

- **Compiler can issue a load prior to a preceding, possibly-conflicting store**



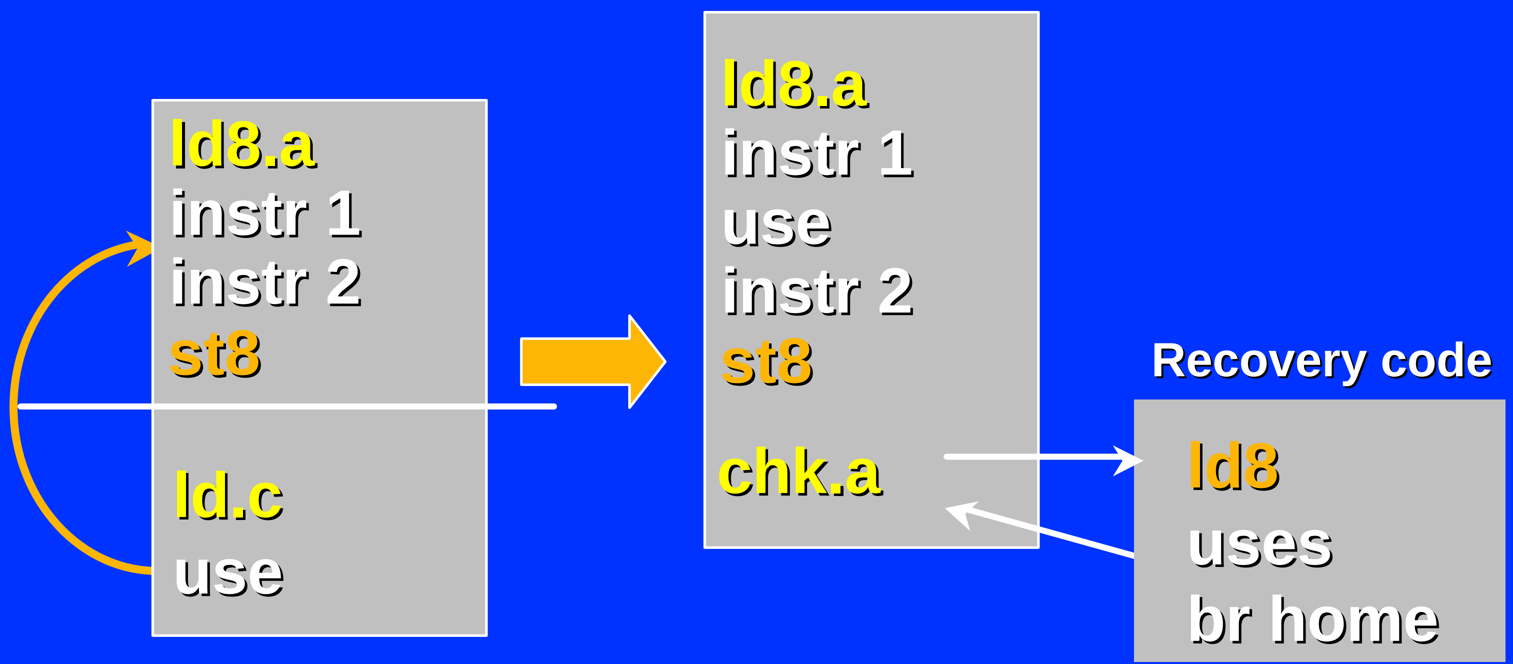
Unique feature to IA-64

intel®

HEWLETT PACKARD

Data Speculation

- Uses can be hoisted



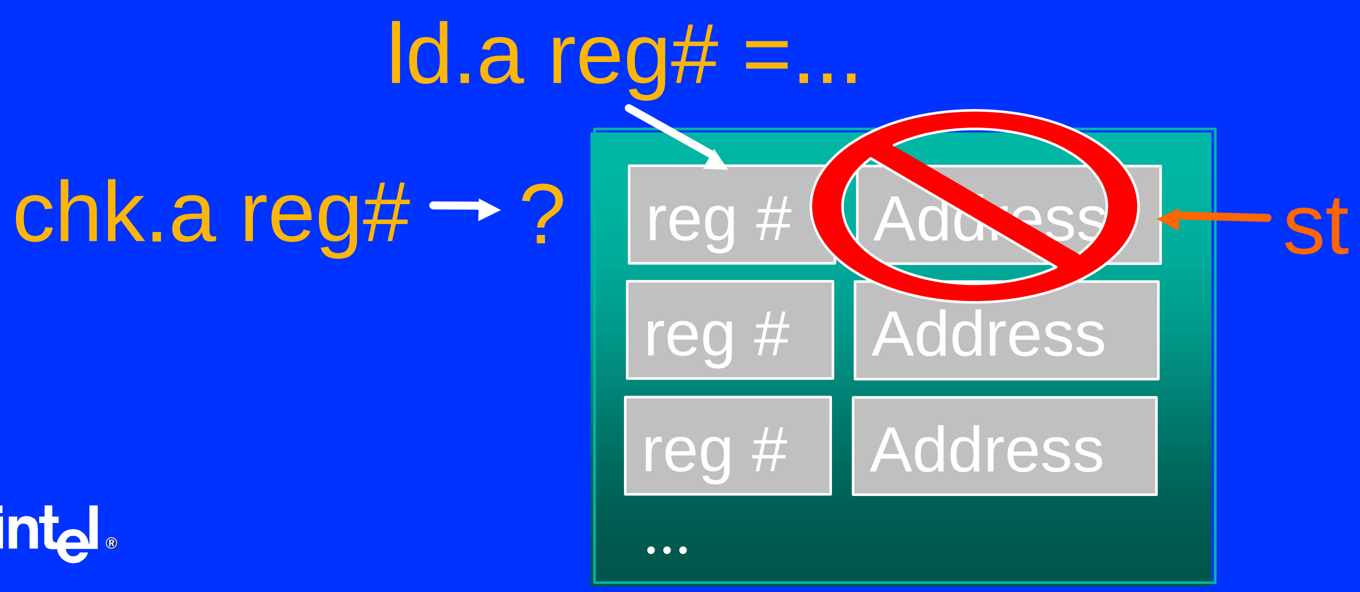
Synergy with control speculation yields greater performance

intel®

HEWLETT PACKARD

Advanced Load Address Table - ALAT

- ld.a inserts entries.
- Conflicting stores remove entries
  - Also: ld.c.clr, chk.a.clr,
- Presence of entry indicates success
  - chk.a branches when no entry is found



Architectural Support for Data Speculation

- Instructions
  - ld.a - advanced loads
  - ld.c - check loads
  - chk.a - advance load checks
- Speculative Advanced loads - ld.sa - is an advanced load with deferral
- ALAT - HW structure containing outstanding advanced loads

Speculation Benefits

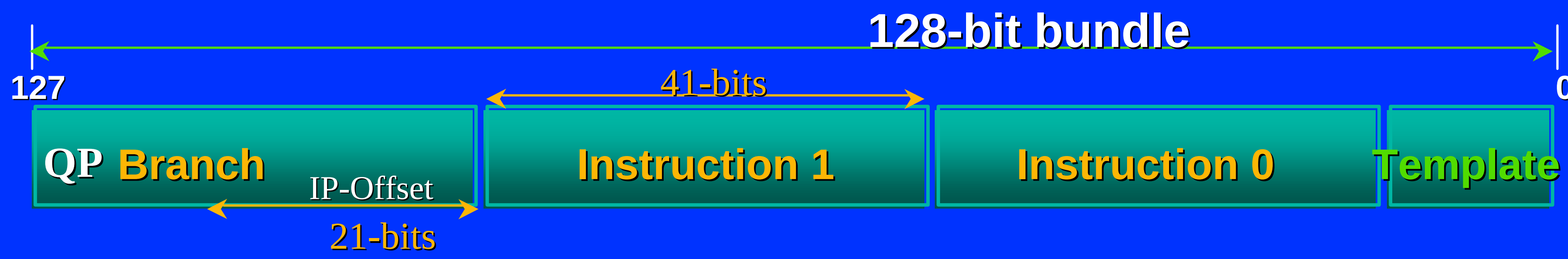
- Reduces impact of memory latency
  - Study demonstrates performance improvement of 79% when combined with predication*
- Greatest improvement to code with many cache accesses
  - Large databases
  - Operating systems
- Scheduling flexibility enables new levels of performance headroom

* August, et.al, 1998

Agenda

- ✓ Architecture Principles
- ✓ Predication & Speculation
- Branch Architecture
- Software Pipelining

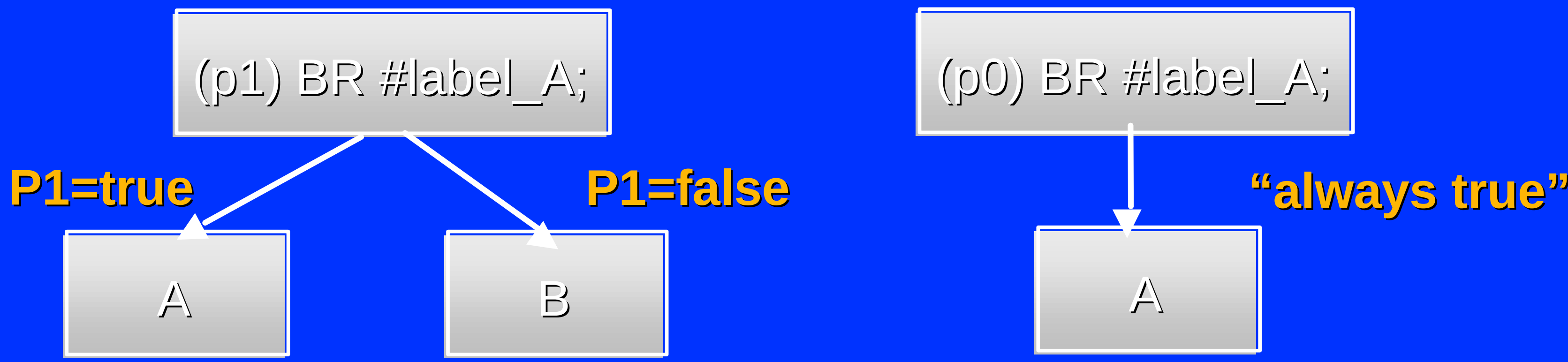
Branch Instruction



- Two basic branch formats
  - Relative: IP := IP + Offset21
  - Indirect: IP := BR[I]
    - 8 branch registers for efficient branch execution
    - Call/Return linking through branch registers
- Loop branches with 64-bit loopcount register (LC)
  - Enables perfect branch prediction of counted loops
  - Traditional architectures always mispredict last iteration
    - Incurs misprediction stall costing many cycles

Branch Predicates

Conditional branches | **Unconditional branches**



- Compiler directed static prediction augments dynamic prediction
  - Better predict highly correlated branches (always/never taken)
  - Frees space in H/W predictor
  - Can give hint for dynamic predictor

Compare & Branch in Same Cycle

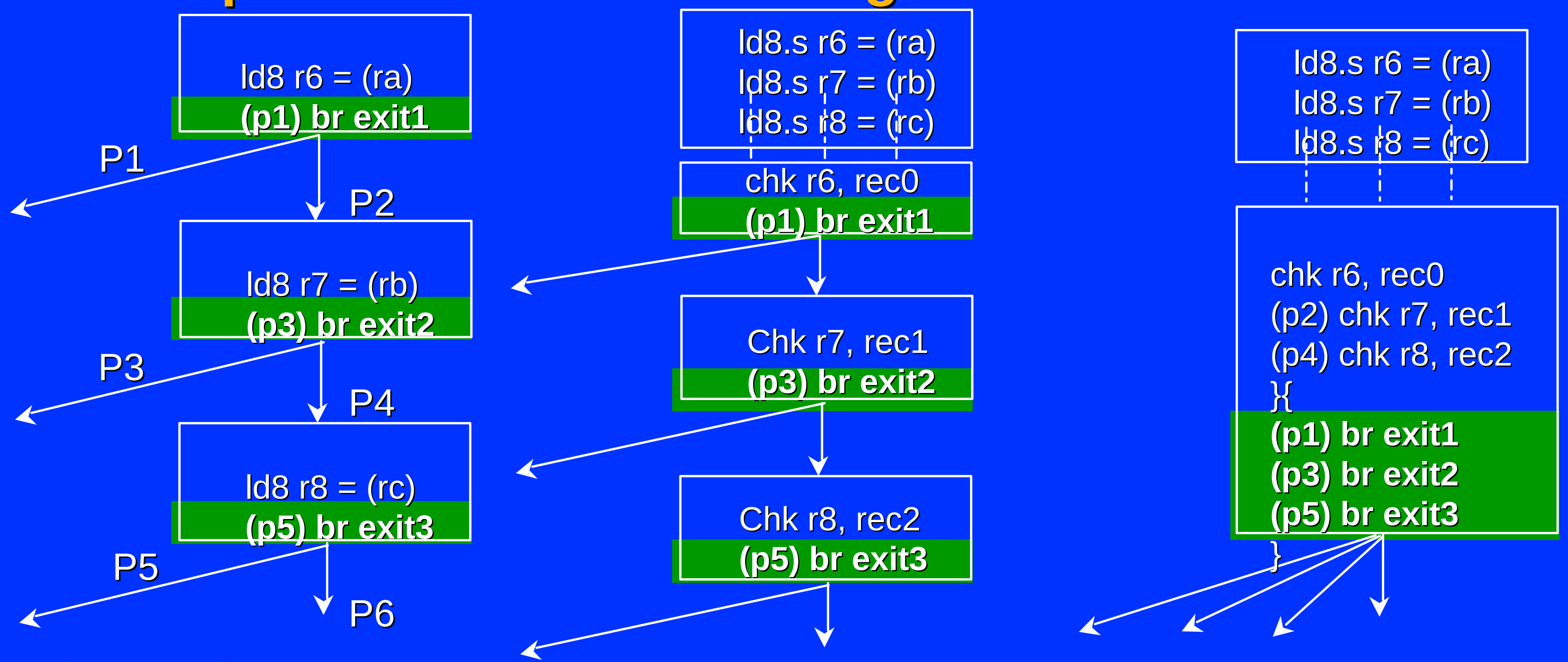
Queens Loop: Parallel Compares & Compare-branch

1	R1=&b[j] **R3=&a[i+j]** **R5=&c[i-j+7]** **p1 <- true**
2	ld R2=[R1] **ld R4=[R3]** **ld R6=[R5]**
4	**p1,p2 <- cmp.and(R2==true)** **p1,p2 <- cmp.and(R4==true)** **p1,p2 <- cmp.and(R6==true)** (p1) br then
	else

From 5 Cycles Down to 4

Multi-way Branch

w/o Speculation | **Hoisting Loads** | **IA-64**



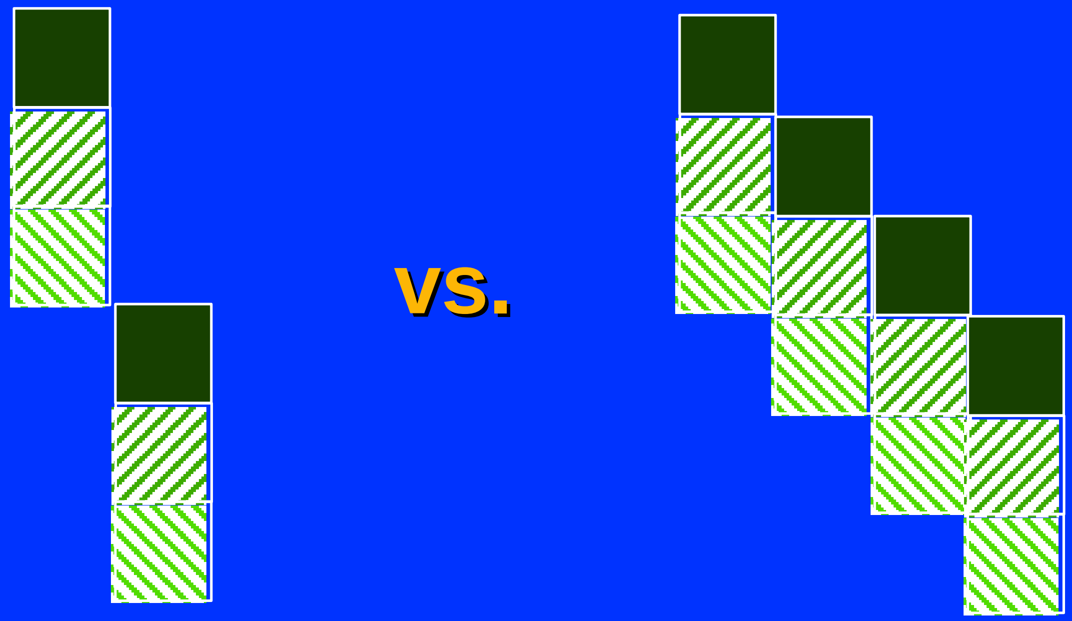
3 branch cycles | **1 branch cycle**

- **Multiway branches: more than 1 branch in a single cycle**
  - **Allows n-way branching**

Supports Aggressive Speculation

Software Pipelining

- Overlapping execution of different loop iterations



- More iterations in same amount of time

Software Pipelining

- IA-64 features that make this possible
  – Full Predication
  – Special branch handling features
  – Register rotation: removes loop copy overhead
  – Predicate rotation: removes prologue & epilogue
- Traditional architectures use loop unrolling
  – High overhead: extra code for loop body, prologue, and epilogue

Especially Useful for Integer Code With Small Number of Loop Iterations

Basic Loop Example

```
For (i=0; i<n; i++) {
    *b++ = *a++;
}   /* MemCopy */
// setup ra/rb/lc,
.label loop
{
 ld8  r35 = [ra],8
}{
 st8  [rb],8 = r35
 br.cloop #loop // check n!=0
}
```

3 ops

Basic Copy Loop

Execution (Cycles)

1	ld_1		
2		st_1	br.cloop
3	ld_2		
4		st_2	br. cloop
5	ld_3		
6		st_3	br. cloop
7	ld_4		
8		st_4	br. cloop

- **Simple Non-overlapping iterations**
  - 2 cycles per iteration
  - 3 operations in loop body

Loop Support: Unrolling

```
Test for loop count 0,1
ld8  r34 = [ra],8

.label loop
 ld8    r35 = [ra],8
 st8  [rb],8 = r34
 br.cle #e-exit
 ld8    r34 = [ra],8
 st8  [rb],8 = r35
 br.cloop #loop

 st8    [rb],8 = r34
 br #thru

.label e-exit
 st8  [rb],8 = r35
.label thru
```

10 ops

Unrolled Copy Loop

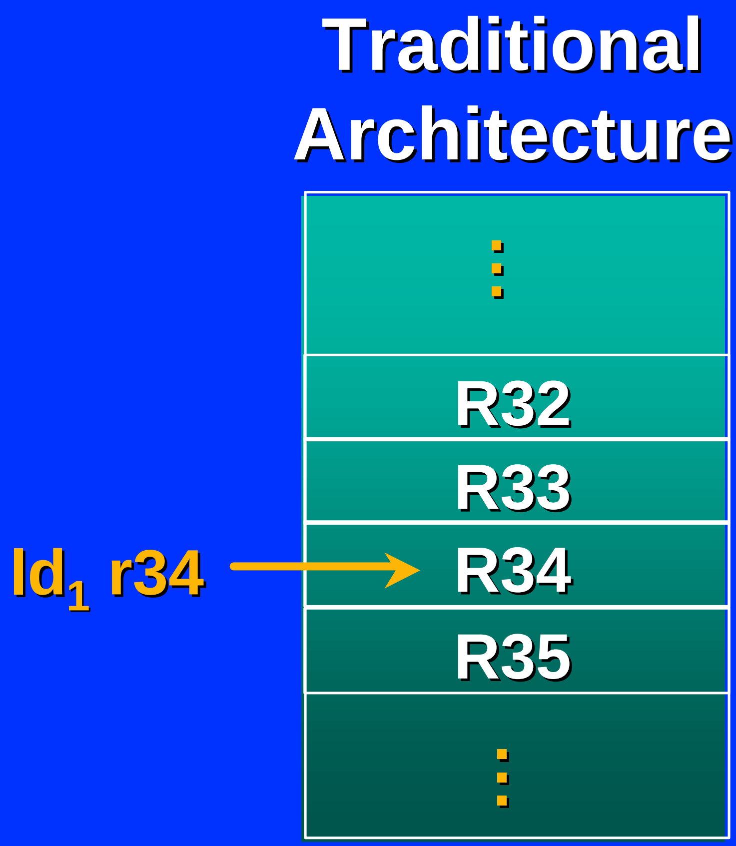
Execution cycles

1	ld_1			**Prologue**
2	ld_2	st_1	br.cle	**Main loop**
3	ld_3	st_2	br.cloop	
4	ld_4	st_3	br.cle	
5		st_4		**Epilogue**

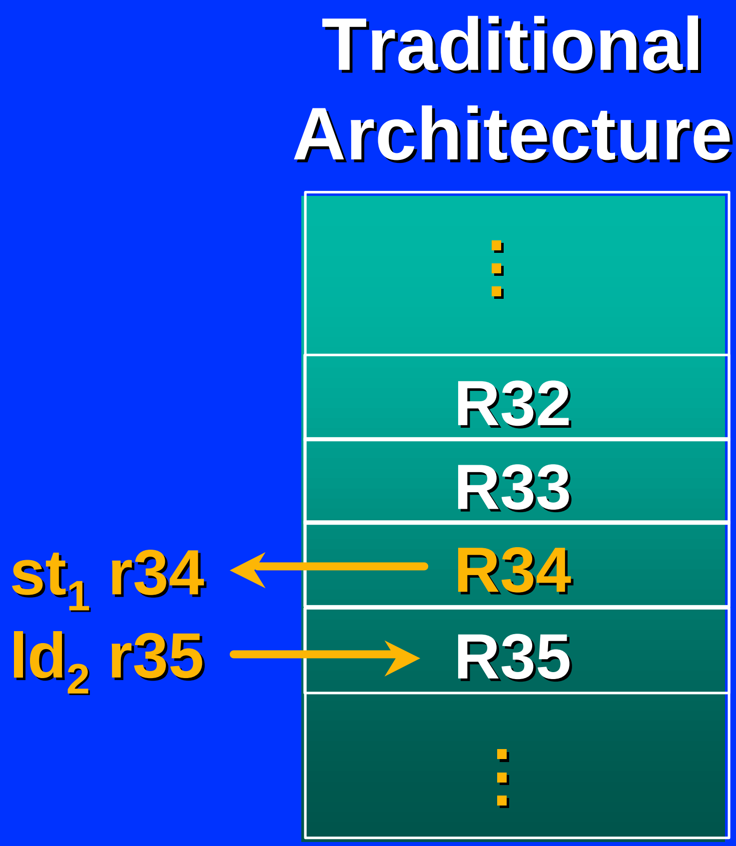
- **Overlapped iterations**
  - **1 cycle per word**
  - **1.6X performance improvement**
  - **3.3X code expansion**

Incurs Code Expansion Penalties

Software Register Renaming

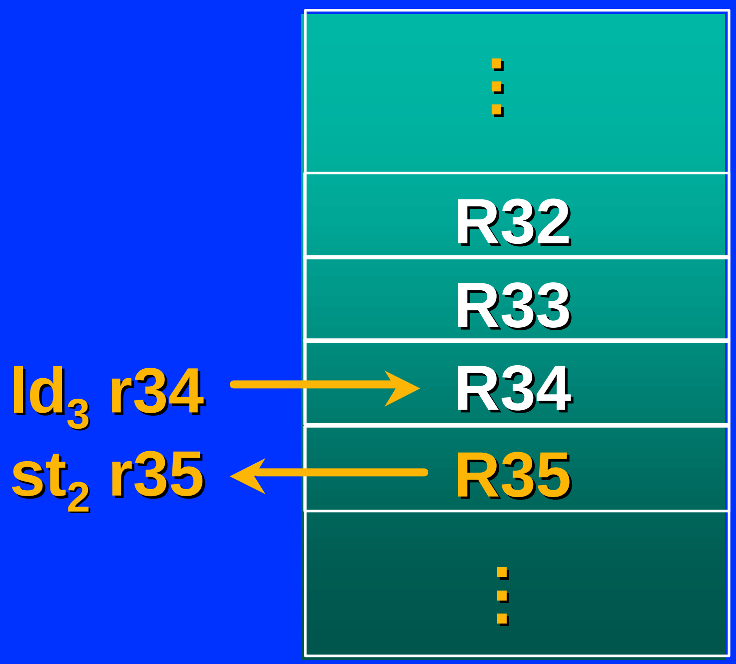


Software Register Renaming



Software Register Renaming

Traditional Architecture



Software Register Renaming

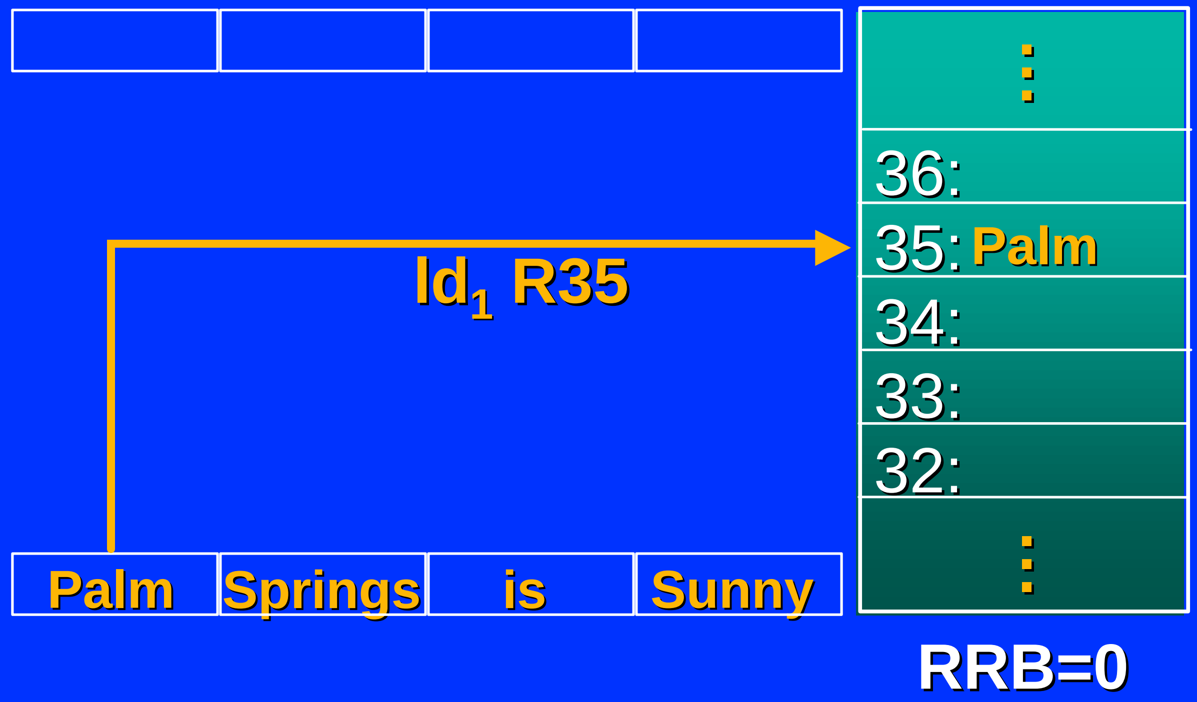
Traditional Architecture

Software Register Renaming

Traditional Architecture

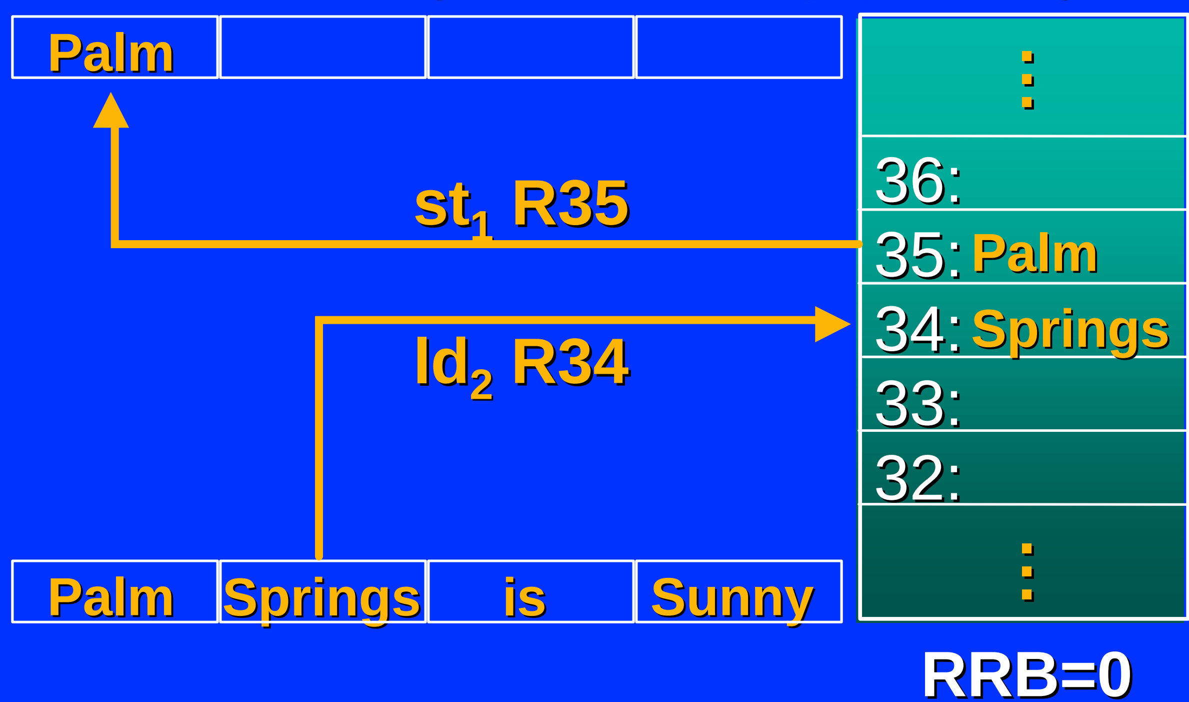
Introducing Rotating Registers

- GR 32-127, FR32-127 can rotate
- Separate Rotating Register Base for each: GRs, FRs
- Loop branches decrement all register rotating bases (RRB)
- Instructions contain a "virtual" register number
  - RRB + virtual register number = physical register number.



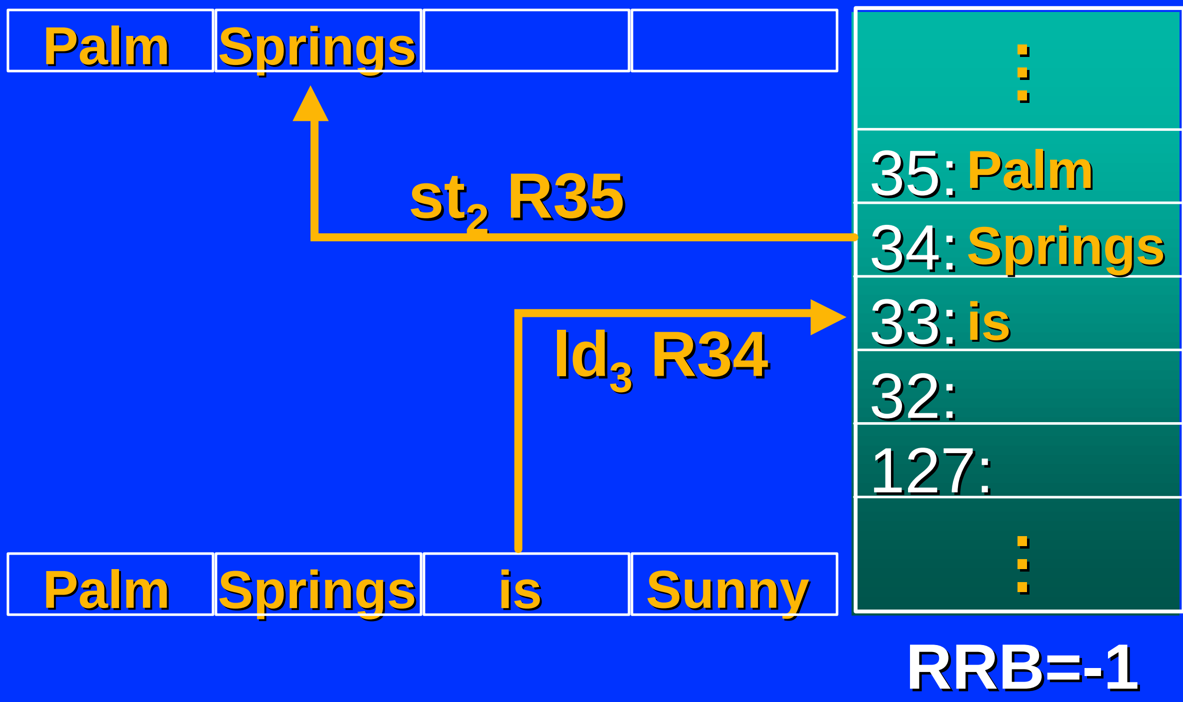
Introducing Rotating Registers

- GR 32-127, FR32-127 can rotate
- Separate Rotating Register Base for each: GRs, FRs
- Loop branches decrement all register rotating bases (RRB)
- Instructions contain a "virtual" register number
  - RRB + virtual register number = physical register number.



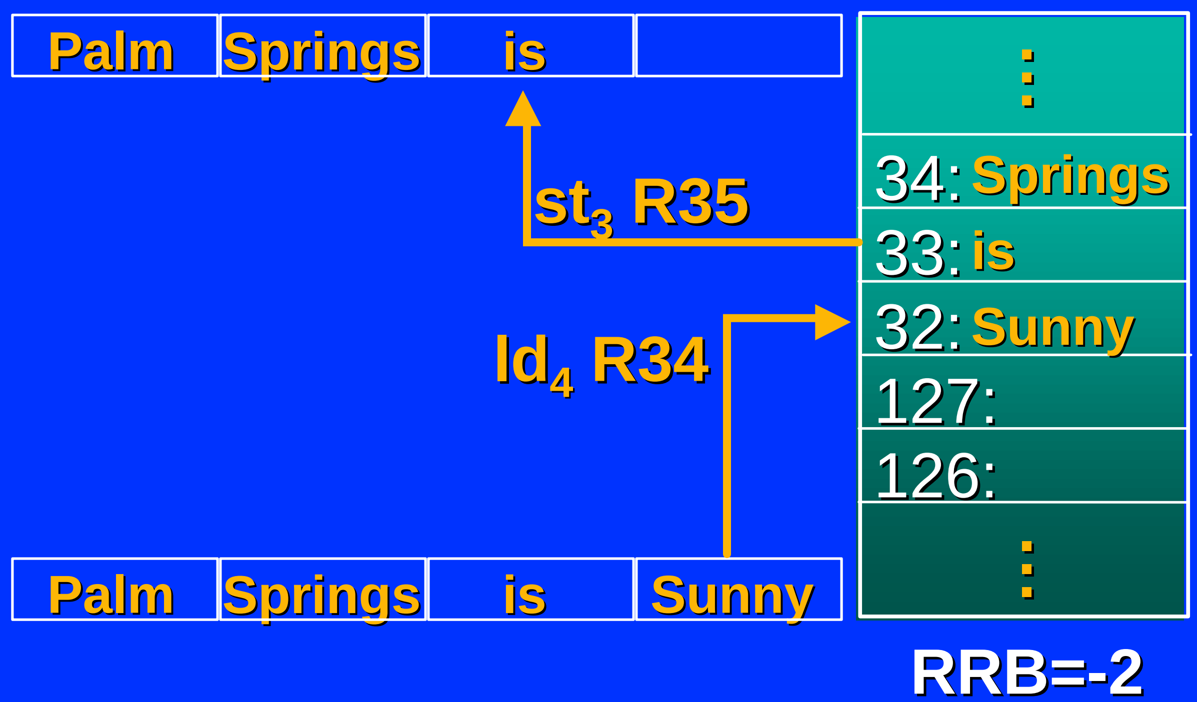
Introducing Rotating Registers

- GR 32-127, FR32-127 can rotate
- Separate Rotating Register Base for each: GRs, FRs
- Loop branches decrement all register rotating bases (RRB)
- Instructions contain a "virtual" register number
  - RRB + virtual register number = physical register number.



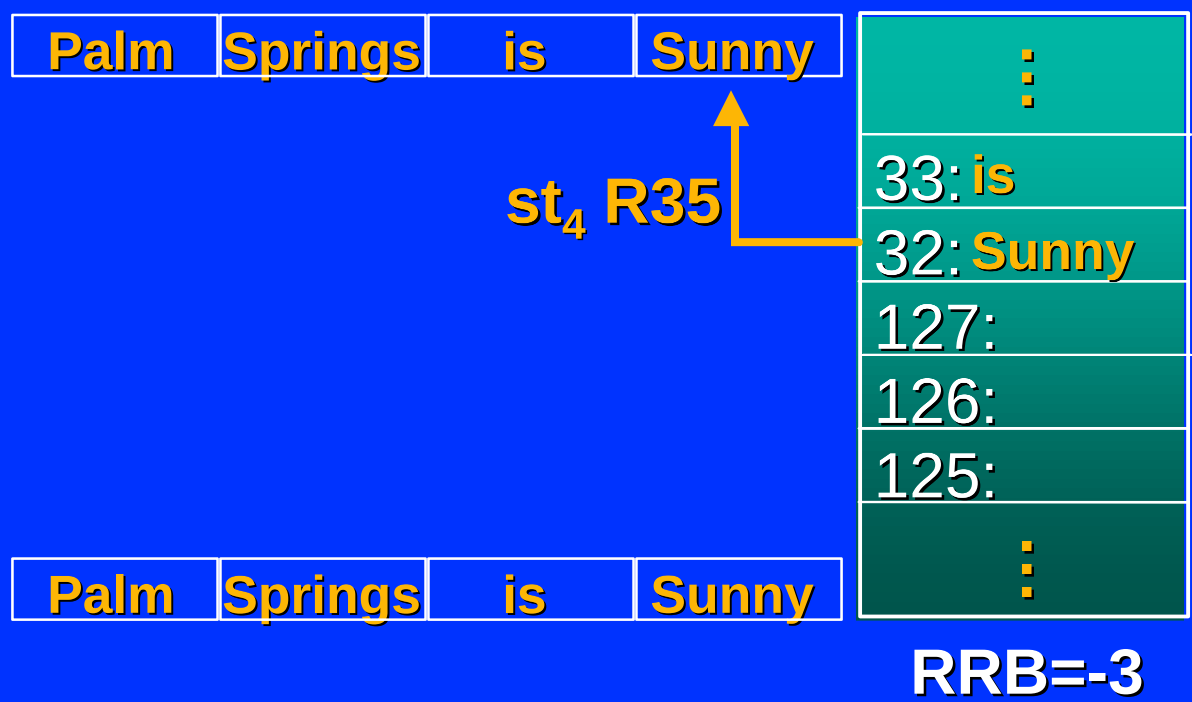
Introducing Rotating Registers

- GR 32-127, FR32-127 can rotate
- Separate Rotating Register Base for each: GRs, FRs
- Loop branches decrement all register rotating bases (RRB)
- Instructions contain a “virtual” register number
  - RRB + virtual register number = physical register number.



Introducing Rotating Registers

- GR 32-127, FR32-127 can rotate
- Separate Rotating Register Base for each: GRs, FRs
- Loop branches decrement all register rotating bases (RRB)
- Instructions contain a “virtual” register number
  - RRB + virtual register number = physical register number.



Loop Support: Rotating Registers

5 ops

```
// setup ra/rb/lc/ec,
check n > 2
{
 ld8  r35 = [ra],8
}
.label loop
{
 ld8   r34 = [ra],8
 st8  [rb] = r35,8
 br.ctop #loop
}{
 st8   [rb] = r35,8
}
```

Software Pipelined Copy Loop

Execution cycles

1	ld_1			Prologue
2	ld_2	st_1	br.ctop	Main loop
3	ld_3	st_2	br. ctop	
4	ld_4	st_3	br. ctop	
5		st_4		Epilogue

- Modulo Scheduled Iterations
  - 1 cycle per word
  - 1.6X performance improvement
    - additional upside for higher latency conditions
  - 1.7X code expansion

Introducing Rotating Predicate Registers

- PR16-63 can rotate, with separate Rotating Register Base
- Loop branches decrement all register rotating base (RRB)
- Instructions contain a "virtual" predicate register number
  - RRB + virtual register number = physical register number.

LC=3
EC=2

⋮	
18:	0
17:	0
16:	1
63:	0
62:	0
⋮	

RRB=0

(p17) st R35
(p16) ld R34

Code

(p16) ld_1 R34 (p17) st R35

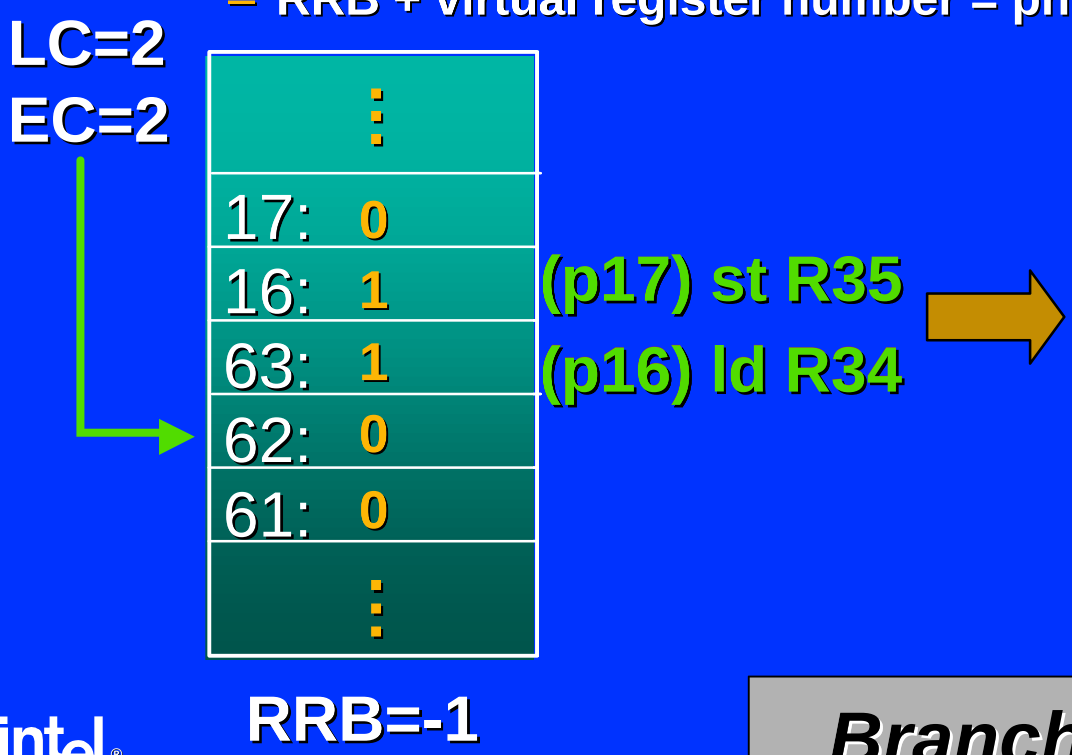
Initialize

intel®

HEWLETT PACKARD

Introducing Rotating Predicate Registers

- PR16-63 can rotate, with separate Rotating Register Base
- Loop branches decrement all register rotating base (RRB)
- Instructions contain a "virtual" predicate register number
  – RRB + virtual register number = physical register number.



Code

(p16) ld_1 R34	**(p17) st R35**
(p16) ld_2 R34	(p17) st_1 R35

Branch 1

intel®

HEWLETT PACKARD

Introducing Rotating Predicate Registers

- PR16-63 can rotate, with separate Rotating Register Base
- Loop branches decrement all register rotating base (RRB)
- Instructions contain a "virtual" predicate register number
  - RRB + virtual register number = physical register number.

LC=1
EC=2

⋮	
16:	1
63:	1
62:	1
61:	0
60:	0
⋮	

RRB=-2

(p17) st R35
(p16) ld R34

Code

(p16) ld_1 R34	(p17) st R35
(p16) ld_2 R34	(p17) st_1 R35
(p16) ld_3 R34	(p17) st_2 R35

Branch 2

intel®

HEWLETT PACKARD

Introducing Rotating Predicate Registers

- PR16-63 can rotate, with separate Rotating Register Base
- Loop branches decrement all register rotating base (RRB)
- Instructions contain a “virtual” predicate register number
  - RRB + virtual register number = physical register number.

LC=0

EC=2

⋮	
63:	1
62:	1
61:	1
60:	0
59:	0
⋮	

RRB=-3

(p17) st R35

(p16) ld R34

Code

(p16) ld_1 R34	**(p17) st R35**
(p16) ld_2 R34	(p17) st_1 R35
(p16) ld_3 R34	(p17) st_2 R35
(p16) ld_4 R34	(p17) st_3 R35

Branch 3

Introducing Rotating Predicate Registers

- PR16-63 can rotate, with separate Rotating Register Base
- Loop branches decrement all register rotating base (RRB)
- Instructions contain a "virtual" predicate register number
  – RRB + virtual register number = physical register number.

LC=0
EC=1

⋮	
62:	1
61:	1
60:	0
59:	0
58:	0
⋮	

RRB=-4

(p17) st R35
(p16) ld R34

Code

(p16) ld_1 R34	(p17) st R35
(p16) ld_2 R34	(p17) st_1 R35
(p16) ld_3 R34	(p17) st_2 R35
(p16) ld_4 R34	(p17) st_3 R35
(p16) ld R34	(p17) st_3 R35

Branch 4

Introducing Rotating Predicate Registers

- PR16-63 can rotate, with separate Rotating Register Base
- Loop branches decrement all register rotating base (RRB)
- Instructions contain a "virtual" predicate register number
  - RRB + virtual register number = physical register number.

LC=0
EC=0

⋮		
61:	1	
60:	0	(p17)
59:	0	(p16)
58:	0	
57:	0	
⋮		

RRB=-5

Code

(p16) ld_1 R34	**(p17) st R35**
(p16) ld_2 R34	(p17) st_1 R35
(p16) ld_3 R34	(p17) st_2 R35
(p16) ld_4 R34	(p17) st_3 R35
(p16) ld R34	(p17) st_4 R35

Fall Through

Fall Through

Loop Support: Rotating Predicates

```
// setup ra/rb/lc/ec,
check n > 1
.label loop
{
 (p16) ld8   r34 = [ra],8
 (p17) st8  [rb] = r35,8
  br.ctop #loop
}
```

3 ops

Software Pipelined Copy Loop

Execution cycles

1	ld_1	st	br.ctop	
2	ld_2	st_1	br. ctop	Main loop
3	ld_3	st_2	br. ctop	
4	ld_4	st_3	br. ctop	
5	ld	st_4	br. ctop	

- **Software Pipelined MemCopy**
  - 1 cycle per word
  - 1.6X performance improvement
  - no code expansion

Efficient Loop, Efficient Code Size

Software Pipelining Benefits

- Loop pipelining maximizes performance; minimizes overhead
  - Avoids code expansion of unrolling and code explosion of prologue and epilogue
  - Smaller code means fewer cache misses
  - Greater performance improvements in higher latency conditions
- Reduced overhead allows S/W pipelining of small loops with unknown trip counts
  - Typical of integer scalar codes

Reviewing What's New:

- Parallel compares
- Tbit
- Nat bits
- Deferral
- Hoisting uses
- Propagation
- Branch instructions
- Static prediction
- Advanced loads
- ALAT
- Loop branches
- LC register
- EC register
- Multiway branch
- Branch registers
- Register rotation
- Predicate rotation
- RRB

Summary

- Speculation reduces memory latency impact
  - IA-64 removes recovery from critical path
  - Benefits applications with poor cache locality: server applications, OS
- Predication removes branches
  - Parallel compares increase parallelism
  - Benefits complex control flow: large databases
- S/W pipelining support with minimal overhead enables broad usage
  - Performance for small integer loops with unknown trip counts as well as monster FP loops